SOFTWAREUPGRADEAPP

Code analysis

By: Gateway

2022-03-29

CONTENT

INTRODUCTION

This document contains results of the code analysis of SoftwareUpgradeApp.

CONFIGURATION

- Quality Profiles
  - Names: Sonar way [Java]; Stryker [Kotlin]; Stryker [XML];
  - Files: AX9JLBXVZWz9exVyjg9p.json; AX_WRsJAZoNvhpLrXfw9.json; AX_WbTE_ZoNvhpLrXf1b.json;
- Quality Gate
  - Name: Sonar-StrykerAndroid
  - File: Sonar-StrykerAndroid.xml

SYNTHESIS

ANALYSIS STATUS

Reliability	Security	Security Review	Maintainability
A	A	A	A

QUALITY GATE STATUS

Quality Gate Status	Passed

Metric	Value
Reliability Rating on New Code	OK
Security Rating on New Code	OK
Maintainability Rating on New Code	OK
Duplicated Lines (%) on New Code	OK

METRICS

Coverage	Duplication	Comment density	Median number of lines of code per file	Adherence to coding standard
0.0 %	0.0 %	7.0 %	32.0	99.8 %

TESTS

Total	Success Rate	Skipped	Errors	Failures
0	0 %	0	0	0

DETAILED TECHNICAL DEBT

Reliability	Security	Maintainability	Total
-	-	0d 7h 17min	**0d 7h 17min**

METRICS RANGE

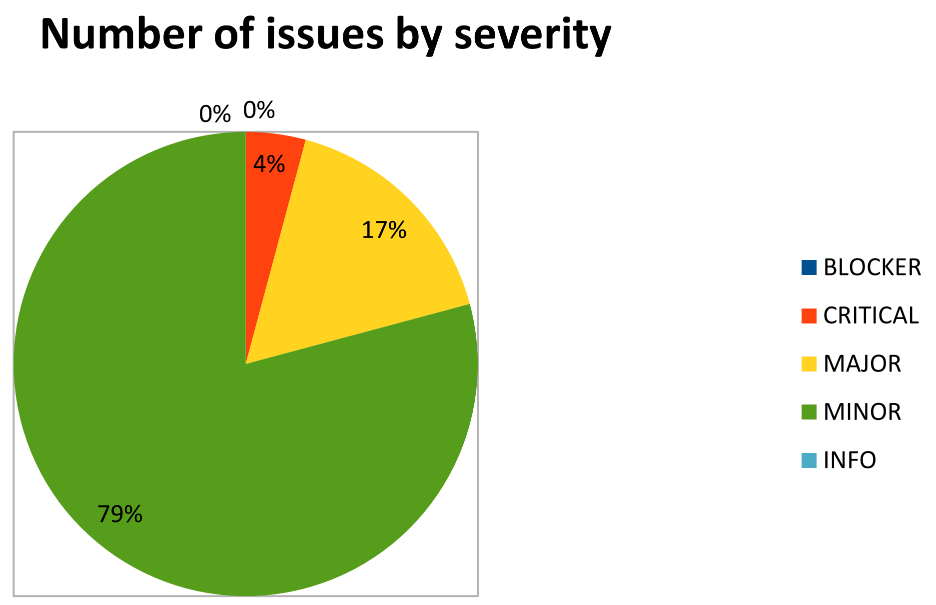
	Cyclomatic Complexity	Cognitive Complexity	Lines of code per file	Comment density (%)	Coverage	Duplication (%)
Min	0.0	0.0	4.0	0.0	0.0	0.0
Max	538.0	624.0	6394.0	25.9	0.0	0.0

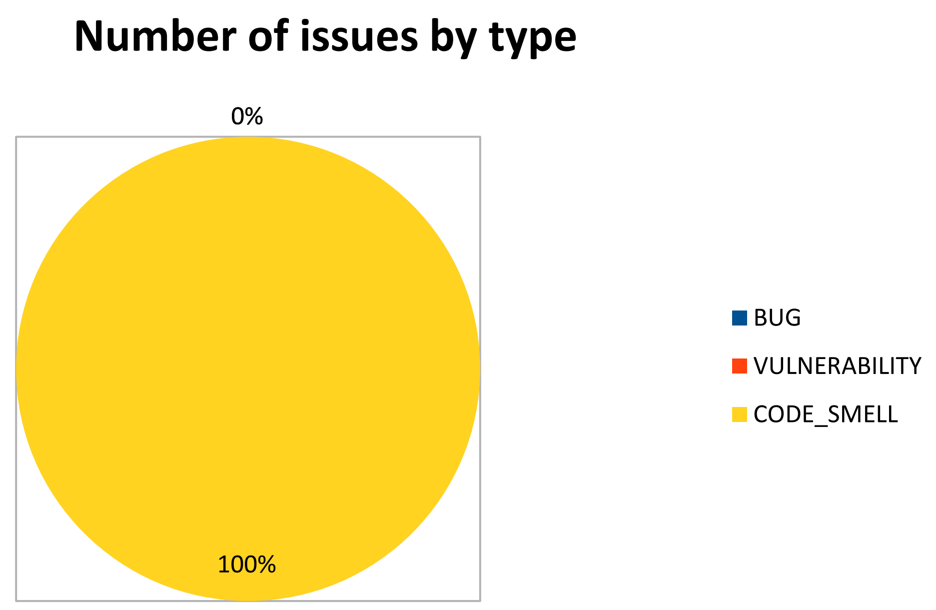
VOLUME

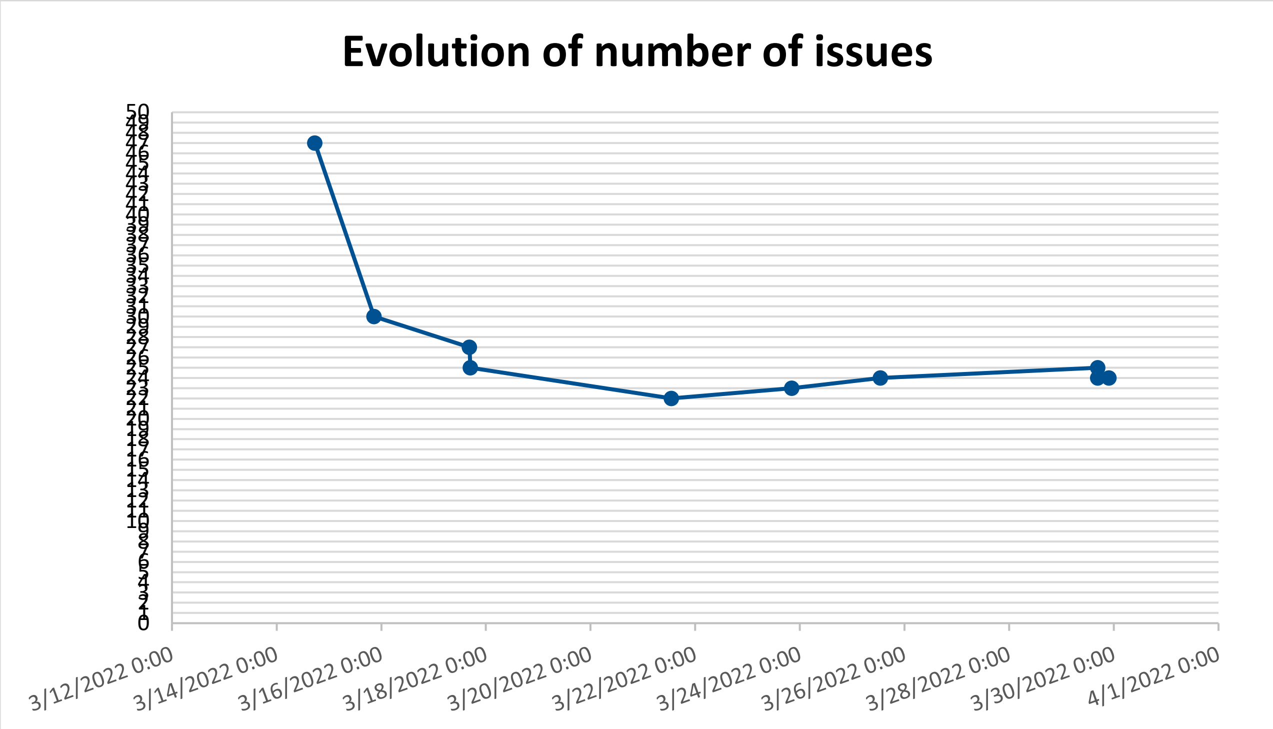
Language	Number
Java	143
Kotlin	4849
XML	1402
Total	6394

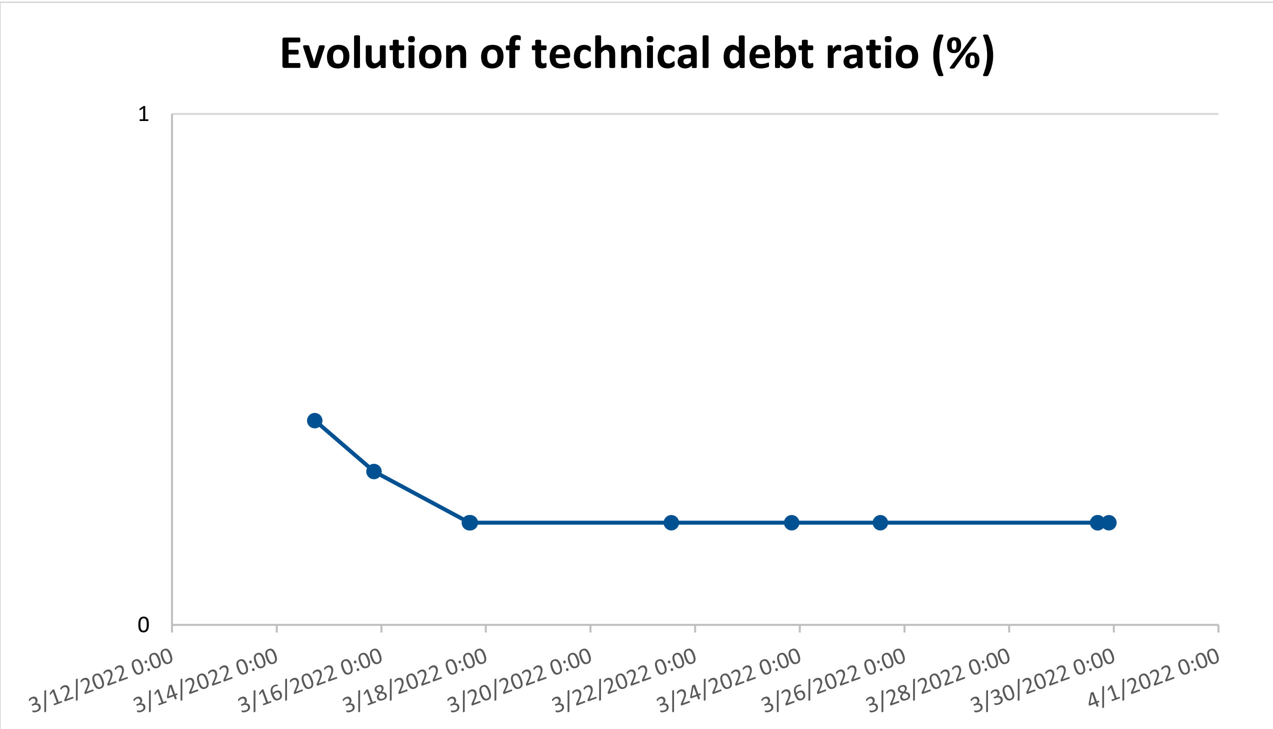
ISSUES

CHARTS






Evolution of number of issues
0
5
10
15
20
25
30
35
40
45
50
3/12/2022 0:00
3/14/2022 0:00
3/16/2022 0:00
3/18/2022 0:00
3/20/2022 0:00
3/22/2022 0:00
3/24/2022 0:00
3/26/2022 0:00
3/28/2022 0:00
3/30/2022 0:00
4/1/2022 0:00


Evolution of technical debt ratio (%)
1
0
3/12/2022 0:00
3/14/2022 0:00
3/16/2022 0:00
3/18/2022 0:00
3/20/2022 0:00
3/22/2022 0:00
3/24/2022 0:00
3/26/2022 0:00
3/28/2022 0:00
3/30/2022 0:00
4/1/2022 0:00

ISSUES COUNT BY SEVERITY AND TYPE

Type / Severity	INFO	MINOR	MAJOR	CRITICAL	BLOCKER
BUG	0	0	0	0	0
VULNERABILITY	0	0	0	0	0
CODE_SMELL	0	19	4	1	0

ISSUES LIST

Name	Description	Type	Severity	Number
Cognitive Complexity of functions should not be too high	Cognitive Complexity is a measure of how hard the control flow of a function is to understand. Functions with high Cognitive Complexity will be difficult to maintain. See Cognitive Complexity	CODE_SMELL	CRITICAL	1
Coroutine usage should adhere to structured concurrency principles	Kotlin coroutines follow the principle of structured concurrency. This helps in preventing resource leaks and ensures that scopes are only exited once all child coroutines have exited. Hence, structured concurrency enables developers to build concurrent applications while having to worry less about cleaning up concurrent tasks manually. It is possible to break this concept of structured concurrency in various ways. Generally, this is not advised, as it can open the door to coroutines being leaked or lost. Ask yourself if breaking structured concurrency here is really necessary for the application's business logic, or if it could be avoided by refactoring parts of the code. This rule raises an issue when it detects that the structured concurrency principles are violated. It avoids reporting on valid use cases and in situations where developers have consciously opted into using delicate APIs (e.g. by using the @OptIn annotation) and hence should be aware of the possible pitfalls. Noncompliant Code Example GlobalScope: fun main() { GlobalScope.launch { // Noncompliant: no explicit opt-in to DelicateCoroutinesApi // Do some work }.join() } Manual job instantiation: fun startLongRunningBackgroundJob(job: Job) { val coroutineScope = CoroutineScope(job) coroutineScope.launch(Job()) { // Noncompliant: new job instance passed to launch() // Do some work } } Manual supervisor instantiation: coroutineScope { launch(SupervisorJob()) { // Noncompliant: new supervisor instance passed to launch() // Do some work } } Compliant Solution In many situations, a good pattern is to	CODE_SMELL	MAJOR	2

	use coroutineScope as provided in suspending functions: suspend fun main() { worker() } suspend fun worker() { coroutineScope { launch { // Compliant: no manually created job/supervisor instance passed to launch() // Do some work } } } GlobalScope: @OptIn(DelicateCoroutinesApi::class) fun main() { GlobalScope.launch { // Compliant: explicit opt-in to DelicateCoroutinesApi via method annotation // Do some work }.join() } No manual job instantiation: fun startLongRunningBackgroundJob(job: Job) { val coroutineScope = CoroutineScope(job) coroutineScope.launch { // Compliant: no manually created job/supervisor instance passed to launch() // Do some work } } Using a supervisor scope instead of manually instantiating a supervisor: supervisorScope { launch { // Do some work } } See Structured concurrency in the Kotlin docs GlobalScope documentation coroutineScope documentation Android coroutines best practices			
Dispatchers should be injectable	Dispatchers should not be hardcoded when using withContext or creating new coroutines using launch or async. Injectable dispatchers ease testing by allowing tests to inject more deterministic dispatchers. You can use default values for the dispatcher constructor arguments to eliminate the need to specify them explicitly in the production caller contexts. This rule raises an issue when it finds a hard-coded dispatcher being used in withContext or when starting new coroutines. Noncompliant Code Example class ExampleClass { suspend fun doSomething() { withContext(Dispatchers.Default) { // Noncompliant: hard-coded dispatcher ... } } } Compliant Solution class ExampleClass(private val dispatcher: CoroutineDispatcher = Dispatchers.Default) { suspend fun doSomething() { withContext(dispatcher) { ... } } } See Inject dispatchers (Android coroutines best practices)	CODE_SMELL	MAJOR	2
Code annotated as deprecated should not be used	Code annotated as deprecated should not be used since it will be removed sooner or later. Noncompliant Code Example @Deprecated("") interface Old class Example : Old // Noncompliant	CODE_SMELL	MINOR	19

SECURITY HOTSPOTS

SECURITY HOTSPOTS COUNT BY CATEGORY AND PRIORITY

Category / Priority	LOW	MEDIUM	HIGH
LDAP Injection	0	0	0
Object Injection	0	0	0
Server-Side Request Forgery (SSRF)	0	0	0
XML External Entity (XXE)	0	0	0
Insecure Configuration	0	0	0
XPath Injection	0	0	0
Authentication	0	0	0
Weak Cryptography	0	0	0
Denial of Service (DoS)	0	0	0
Log Injection	0	0	0
Cross-Site Request Forgery (CSRF)	0	0	0
Open Redirect	0	0	0
SQL Injection	0	0	0
Buffer Overflow	0	0	0
File Manipulation	0	0	0
Code Injection (RCE)	0	0	0
Cross-Site Scripting (XSS)	0	0	0
Command Injection	0	0	0
Path Traversal Injection	0	0	0

HTTP Response Splitting	0	0	0
Others	0	0	0

SECURITY HOTSPOTS LIST